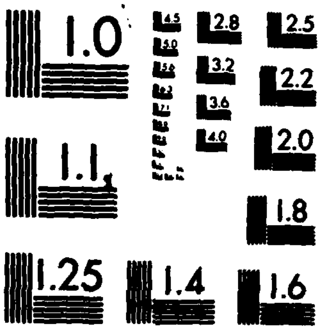
AD-A129 665 WORST-CASE ANALYSES OF SELF-ORGANIZING SEQUENTIAL 1/1
SEARCH HEURISTICS..(U) CARNEGIE-MELLON UNIV PITTSBURGH
PA DEPT OF COMPUTER SCIENCE J L BENTLEY ET AL.
UNCLASSIFIED 28 MAR 83 CMU-CS-83-121 N00014-76-C-0370 F/G 12/1 NL

END
7-83


1.0
4.5
5.0
5.6
6.3
7.1
2.8
3.2
3.6
4.0
2.5
2.2
2.0
1.1
1.8
1.25
1.4
1.6

CMU-CS-83-121

15

Worst-Case Analyses of Self-Organizing Sequential Search Heuristics[1]

Jon Louis Bentley[2]
Catherine Cole McGeoch
Department of Computer Science
Carnegie-Mellon University
Pittsburgh, Pennsylvania 15213

28 March 1983

DEPARTMENT
of
COMPUTER SCIENCE

Carnegie-Mellon University

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER CMU-CS-83-121	2. GOVT ACCESSION NO. AD-A129 665	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) WORST-CASE ANALYSES OF SELF-ORGANIZING SEQUENTIAL SEARCH HEURISTICS		5. TYPE OF REPORT & PERIOD COVERED Interim
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Jon Louis Bentley- on leave at Bell Labs.,NJ Catherine Cole McGeoch		8. CONTRACT OR GRANT NUMBER(s) N00014-76-C-0370
9. PERFORMING ORGANIZATION NAME AND ADDRESS Carnegie-Mellon University Computer Science Department Pittsburgh, PA 15213		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Office of Naval Research Arlington, VA 22217		12. REPORT DATE March 28, 1983
		13. NUMBER OF PAGES 16
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

The performance of sequential search can be enhanced by the use of heuristics that move elements closer to the front of the list as they are found. Previous analyses have characterized the performance of such heuristics probabilistically. In this paper we show that the heuristics can also be analyzed in the worst-case sense, and

that the relative merit of the heuristics under this analysis is different than in the probabilistic analyses. Simulations show that the relative merit of the heuristics on real data is closer to that of the new worst-case analyses rather than that of the previous probabilistic analyses.

CMU-CS-83-121

-a-

Worst-Case Analyses of Self-Organizing Sequential Search Heuristics[1]

Jon Louis Bentley[2]
Catherine Cole McGeoch
Department of Computer Science
Carnegie-Mellon University
Pittsburgh, Pennsylvania 15213

28 March 1983

Abstract

The performance of sequential search can be enhanced by the use of heuristics that move elements closer to the front of the list as they are found. Previous analyses have characterized the performance of such heuristics probabilistically. In this paper we show that the heuristics can also be analyzed in the worst-case sense, and that the relative merit of the heuristics under this analysis is different than in the probabilistic analyses. Simulations show that the relative merit of the heuristics on real data is closer to that of the new worst-case analyses rather than that of the previous probabilistic analyses.

[1] This research was supported in part by the Office of Naval Research under contract N00014-76-C-0370 and in part by the National Science Foundation under an NSF Graduate Fellowship.

[2] Currently on leave at Bell Telephone Laboratories, Murray Hill, New Jersey, 07974.

Table of Contents

1. Introduction

The performance of sequential search in an unsorted list can be enhanced by the use of *self-organizing* heuristics that attempt to ensure that frequently accessed keys are near the front of the list.[3] The following three heuristics are representative of a larger class.

- **Transpose.** When the key is found, move it one closer to the front of the list by transposing it with the key immediately in front of it.

- **Move-to-Front.** When the key is found, move it to the front of the list (all other keys retain their relative order).

- **Count.** When the key is found, increment its count field (an integer that is initially zero) and move it forward as little as needed to keep the list sorted in decreasing order by count.

Note that the first two heuristics require no memory other than that for representing the lists, while the third heuristic requires an additional count field; the first two heuristics will therefore be called *memoryless.* Previous work (described in the next section) has shown that under various probabilistic assumptions, these heuristics can significantly reduce the time required by sequential search.

In this paper we will investigate the heuristics from a novel viewpoint: that of their *worst-case* performance rather than their *expected* performance. Such an analysis is trivial and uninteresting if we consider the worst-case cost of a single search. We will therefore count the worst-case number of comparisons made by the heuristics for any particular sequence of search keys, and show that for the Move-to-Front and Count heuristics, that number is at most twice the number of comparisons made when using the Optimal Static Ordering (defined in the next section) for the sequence of requests. This result immediately implies the strongest general theorem known for the expected time of the Move-to-Front heuristic. We also give a counterexample that shows that the Transpose heuristic has very poor worst-case performance.

These analyses are of both theoretical and practical interest. This paper emphasizes the theoretical tools used in the worst-case analyses. Our results provide a simpler proof of a stronger theorem regarding an artifact that has been extensively studied for almost two decades. Furthermore, the analyses use a simple but elegant bookkeeping technique of general interest. The practical contribution of this paper is not so much prescriptive as descriptive: practitioners have long used the Move-to-Front heuristic even though theoretical analyses indicated that the Transpose heuristic was superior. Our analysis provides a metric under which Move-To-Front is superior to Transpose and thereby explains the actions of the practitioners.

[3] By the term list we refer only to its sequential nature; our results apply to lists implemented with arrays or with records and pointers.

This paper is organized in six sections. An overview of previous work can be found in Section 2. With that background, the new results are presented in Section 3. Section 4 describes the results of empirical studies, and advice to practitioners is offered in Section 5. The paper is summarized in Section 6.

2. Previous Work

In this section we will survey previous results concerning self-organizing heuristics. Only the more important results are are presented here; for further study, consult the references at the end of the paper.

The heuristics (or rules) that we study deal with searches for elements of a set of N keys stored in a list. A particular query is answered by performing a sequential search for the requested key and then reordering the list according to some search rule. A string of requests forms a *request sequence*. An important kind of request sequence independently chooses the i^{th} element with probability p_i according to the probability distribution $P = (p_1, p_2, \ldots, p_N)$. The *cost* of a search rule for such a distribution has been defined as the asymptotic expected search cost for a single key (measured as the number of comparisons made) when the set is being reordered according to the rule; we will denote this cost by $A_R(P)$ for rule R. The *Optimal Static Ordering* for the set is one in which the keys are arranged in decreasing order of request probabilities and never reordered. While this is not necessarily optimal over all rules (because it is static, rather than dynamic), it is used as a basis for comparing the performance of the heuristics, and its cost will be denoted by $A_O(P)$. The heuristics have been studied under this asymptotic model since 1965; we present the significant results below. The Count heuristic is considered after the two memoryless heuristics.

The asymptotic expected search cost $A_M(P)$ under the Move-to-Front rule for the probability distribution P has been given by McCabe [1965], Burville and Kingman [1973], Knuth [1973], Hendricks [1976], Rivest [1976], and Bitner [1979]. It is known that for any distribution P, the cost $A_M(P)$ is at most twice the cost of the Optimal Static Ordering, $A_O(P)$. The asymptotic expected cost for Transpose, $A_T(P)$, was shown by Rivest [1976] to be less than or equal to $A_M(P)$; this bound is strict for all P except where all the nonzero probabilities are equal or when $N=2$.

Rivest defined the *optimal permutation rule* to be one with least cost for all P and any initial ordering of the keys, and conjectured that Transpose is optimal. Yao (reported in Bitner [1976]) and Bitner [1982] have given distributions where Transpose is optimal over all rules, but Anderson, Nash, and Weber [1982] presented a counterexample to the conjecture by finding a rule that is better than Transpose for a specific distribution. Recent work has examined modifications of the heuristics and asymptotic costs for classes of probability distributions: see Gonnet, Munro and Suwanda [1979], Hendricks [1973], Bitner [1976, 1982], Kan and Ross [1981], and Tenenbaum and Nemes [1982]. Zipf's Law is a natural distribution; Knuth [1973] showed that the cost of Move-to-Front under this distribution is bounded by 2 ln 2 (about 1.386) times the cost of the Optimal Static Ordering. The highest ratio yet found is $\pi/2$ (by Gonnet, Munro, and Suwanda [1979]), and the existence of a bound tighter than $2 \cdot A_O(P)$ remains an open problem.

Measurements other than asymptotic cost have been considered. Bitner [1976, 1979] showed that while Transpose is asymptotically more efficient, Move-to-Front converges more quickly. Therefore, Move-to-Front is preferred when the number of requests is not large. He proposed a *hybrid rule*, which changes from Move-to-Front to Transpose when the number of requests falls in a certain range: he suggests from $\Theta(N)$ to $\Theta(N^2)$ requests as the change point for Zipf's Law. Bitner also discussed the *overwork* (the area between the cost curve and its asymptote) for the two rules, and presented distributions for which Move-to-Front performs much better than Transpose under this measure. Under Zipf's Law, for instance, the overwork is $O(N^2)$ for Move-to-Front and is $\Omega(N^3)$ for Transpose.

Rivest [1976] introduced a range of *move-ahead-k* heuristics, where a requested key is moved ahead k positions ($k=1$ corresponds to Transpose and $k=N-1$ corresponds to Move-to-Front), and simulated the asymptotic behavior of these heuristics for values of k from 1 to 6 and values of N from 3 to 12, for 5000 requests distributed by Zipf's Law. On the basis of those results, Bitner [1976] conjectured that for any two heuristics in this range, one will converge faster and the other will have lower asymptotic cost; Gonnet, Munro, and Suwanda [1979] later proved this. Tenenbaum [1978] performed similar tests for N from 3 to 230 and for 12,000 requests, with k from 1 to 7. His results indicate that for larger N and this number of requests a heuristic other than Transpose is more efficient.

The Count heuristic introduces a frequency count f_i of requests for the i^{th} key. Because of this extra information, Count has not been considered to be in the same class as the first two heuristics and has received less attention. By the law of large numbers, if $p_i > p_j$, then the frequency f_i may be less than f_j for only a finite number of requests. The search cost under Count therefore asymptotically approaches that of the Optimal Static Ordering. Bitner [1976] showed that if P is not known beforehand, then Count is at all times optimal.

Various modifications of Count have been proposed to reduce the space needed to maintain the frequency counts. Bitner [1976, 1979] suggested that it is better to maintain the differences between frequencies of adjacent keys rather than their actual counts and also proposed a *limited-difference* rule, where the counts are left unchanged after some upper limit is reached. Other modifications have been suggested for use in combination with Move-to-Front or Transpose. Gonnet, Munro and Suwanda [1979], and Kan and Ross [1980] have examined *k-in-a-row* heuristics, where a key is moved only after it has been requested k times in a row, and Bitner [1976] has analyzed rules of the form *wait-c-and-move*. Lam, Sui, and Yu [1981] presented a scheme that was shown to be optimal over all heuristics that use frequency information.

3. Worst-Case Analyses

The primary results of the previous section can be summarized as follows: for any probability distribution P,

$$A_M(P) \leq 2 \cdot A_O(P),$$
$$A_T(P) \leq A_M(P), \text{ and}$$
$$A_C(P) = A_O(P).$$

These results all deal with the asymptotic expected cost of a single search when the queries are from a distribution P. In this section we will take an alternative view by considering the worst-case cost of performing all searches in a given *sequence* of queries S. When the list is being reordered by rule R, we will denote the total number of comparisons required for the sequence S by $C_R(S)$ (signifying the concrete cost as opposed to the asymptotic cost). The next two subsections will show that for any sequence S,

$$C_M(S) \leq 2 \cdot C_O(S), \text{ and}$$
$$C_C(S) \leq 2 \cdot C_O(S).$$

In Subsection 3.3 we will show that such a result does not hold for the Transpose heuristic by exhibiting a particular sequence with very poor performance under that rule.

Before describing the results we must define precisely our cost functions. For the Move-to-Front, Count, and Transpose rules we define $C_R(S)$ (the cost of rule R on the request sequence S) by considering the effect of S on an initially empty search list. For each element t of S we in turn search the current list of size m at a cost of i comparisons if t is in position i or m comparisons if t is not present (in which case we then insert t at the end of the list). In either case we reorder the list by rule R. The cost $C_O(S)$ of the Optimal Static Ordering is fundamentally different: rather than starting with an initially empty list, each search uses the (unchanging) list in which the keys are arranged in decreasing frequency of their counts in S. Note that this assumes that all keys are known in advance, and implies that each search will be successful. The cost of finding an element in position i is i comparisons.

3.1. *Move-To-Front Heuristic*

In this subsection we will show that for any particular sequence of requests S, the number of comparisons made by the Move-To-Front heuristic is never more than twice the number made under the Optimal Static Ordering. To do this, we will reduce the problem to the case in which the request list contains just two distinct keys, analyze that simple case, and finally combine several facts to complete the proof.

The total number of comparisons made for a given request sequence can be divided into two kinds of comparisons: *intraword* comparisons (successfully) compare equal keys, and *interword* comparisons (unsuccessfully) compare unequal keys. For any sequence, the number of intraword comparisons is invariant under all heuristics. For the Move-To-Front heuristic, the total number of interword comparisons is the sum over all distinct pairs of keys of the number of interword comparisons made between each pair. Furthermore, for any sequence S and all pairs of keys A and B, the number of interword comparisons of A to B is exactly the number made for the subsequence of S consisting solely of A's and B's. We call this the *pairwise independence property* of the Move-To-Front heuristic; the number of comparisons made is dependent only on the relative ordering of the A's and B's in the sequence and is independent of other keys. The proof of the property is obvious: accessing an A will cause an (A,B) interword comparison if B is in front of A in the search list, which is true if and only if the last B was accessed more recently than the last A. The other keys in the request sequence do not affect this relationship.

We will now demonstrate the following fact.

Fact 1.

The total number of interword comparisons made by the Move-To-Front heuristic on a sequence of A's and B's is at most twice the number of interword comparisons made by the Optimal Static Ordering applied to the same sequence.

To prove the fact we will assume that the sequence S consists of m A's and n B's, where (without loss of generality), $m \leq n$. Under the Optimal Static Ordering, a total of m interword comparisons will be made (because the search list is always in the order B A, and so only requests for A will cause an interword comparison). Under the Move-To-Front rule, an interword comparison will be made whenever the request sequence changes from an A to a B or from a B to an A. The total number of such changes possible is just twice the number of occurrences of A's (for each change involves an A, and each A can be involved in at most two changes). We therefore know that the total number of comparisons made by Move-To-Front is at most $2m$. Fact 1 follows immediately.

We are now ready to prove the key fact of this subsection.

Fact 2.

For any sequence S, $C_M(S) \leq 2 \cdot C_O(S)$.

We will prove this by simple algebra on the relations

$$C_M(S) = Intra(S) + Inter_M(S) \text{ and}$$
$$C_O(S) = Intra(S) + Inter_O(S),$$

where *Intra* and $Inter_R$ refer to the total number of comparisons of each type made by rule R. By Fact 1 we know that each pair of keys satisfies the factor of two inequality; summing over that inequality for all distinct pairs gives

$$Inter_M(S) \leq 2 \cdot Inter_O(S),$$

for any sequence S. Combining this inequality with the above definition gives Fact 2.

The factor of two in Fact 2 cannot be tightened; the request sequence

$$\mathrm{A\ B\ C\ D\ (D\ C\ B\ A)}^m$$

has $C_O(S) \sim 2m$ but $C_M(S) \sim 4m$.

Knuth [1973, Exercise 6.1-11] shows that $A_M(P) \leq 2 \cdot A_O(P)$ for any distribution P; that exercise is rated M30, implying that it is mathematically oriented and may require over two hours' work to solve. Fact 2 allows us to prove that result easily: we let the sequence S be an arbitrarily long sequence chosen from the distribution P. By Fact 2, we know that

$$C_M(S) \leq 2 \cdot C_O(S).$$

Let $C_{AOP}(S)$ denote the cost of applying the (asymptotically) optimal ordering for distribution P to the sequence S; because the Optimal Static Order for the sequence is optimal over all static orders, we know

$$C_O(S) \leq C_{AOP}(S).$$

These inequalities combine to show that

$$C_M(S) \leq 2 \cdot C_{AOP}(S).$$

The law of large numbers establishes that for an arbitrarily long sequence S,

$$C_{AOP}(S) / |S| \sim A_O(P),$$

(exactly as in the previously mentioned analysis of the Count heuristic). By definition, we know that

$$C_M(S) / |S| \sim A_M(P).$$

Combining these asymptotic facts with the third inequality yields the desired result.

3.2. Count Heuristic

In this subsection we will show that the cost of the Count heuristic on any particular sequence is at most twice the cost of the Optimal Static Ordering. Because the flow of this subsection is exactly the same as the previous subsection, we will proceed at a faster rate.

The first fact that we must establish is that the Count heuristic has the pairwise independence property: for any sequence S, the number of interword comparisons of A and B is exactly the number made for the subsequence of S consisting solely of A's and B's. This is easily proved: A will precede B in the search list if and only if A has a count greater than B's count, or in the case of equal counts, if A's count was most recently greater than B's. In either case, the positions are not affected by other keys. As in the Move-To-Front heuristic, this pairwise independence allows us to focus on two-element sequences. We therefore prove the following fact.

Fact 3.

The total number of interword comparisons made by the Count heuristic on a sequence of A's and B's is at most twice the number of interword comparisons made by the Optimal Static Ordering applied to the same sequence.

To prove this fact, we again assume that the sequence S consists solely of m A's and n B's, with $m \leq n$. Under the Count heuristic, an interword comparison is made every time the second key in the search list is requested; at that time, its count field is incremented. The count field of A can be incremented while it is in the rear at most m times (because it is requested m times). Furthermore, the count field of B can be incremented while it is in the rear at most m times (because after that it has been requested more than m times and can no longer be in the rear). The number of interword comparisons, then, is bounded by m (requests for A's) plus m (for B's), or $2m$, which is twice the number of comparisons made by the Optimal Static Ordering.

The key fact of this subsection follows from the same kind of reasoning used to establish Fact 2 of the previous subsection.

Fact 4.

For any sequence S, $C_C(S) \leq 2 \cdot C_O(S)$.

Again, the reasoning involves summing over the factor of two inequality. By an example similar to that in the previous section, the factor of two cannot be tightened.

3.3. Transpose Heuristic

In this subsection we will demonstrate that the worst-case ratio of the performance of Transpose to that of the Optimal Static Ordering cannot be bounded by any constant. This is easily observed if we consider the request sequence

$$\mathrm{A\,B\,C\,D\,E\,(E\,D)}^k$$

After the first five elements are stored by Transpose, the sequence of (E D) request pairs will cause those two elements to swap position at the back of the list, and neither will advance. The average cost of a search in this sequence will therefore approach 5, while under the Optimal Static Ordering 1.5 comparisons would suffice. For increasing k, this example gives

$$C_T(S) > 3.33 C_O(S).$$

The constant 3.33 can be increased to $\sim 2k/3$ by increasing the length of the "filler" sequence preceding the "active" pair to $k-2$. Note that this counterexample exploits the fact that the Transpose heuristic does not have the pairwise independence property: the relative order of any two keys depends not only on the request sequence but also on whether the keys are adjacent in the search list.

4. Empirical Results

The theoretical analyses in Sections 2 and 3 are by no means unanimous in their evaluation of the heuristics. To gain further insight into the behavior of the heuristics, we used each to perform word counts on a variety of files; that is, the words in each file served as a request sequence. As each word (defined to be a lower-case alphanumeric string delimited by spaces or punctuation marks, which are ignored) was requested, a linear search of the key list was performed, the count field for the key was incremented, and the list was reordered according to the appropriate rule. Each trial started with an initially empty key list; at the first request for a word, the list was searched to the end to determine its absence and then the reordering occurred as if the element had been found in the (new) last position. Although this application clearly suggests the Count heuristic (since the frequencies must be stored anyway), this type of input is one indicator of the behavior of the heuristics under natural conditions.

The average search cost (defined as the total number of interword comparisons divided by the number of requests) required by each heuristic for each file is reported in Table 1; the best performance for each file is underlined. Under a uniform distribution of request frequencies the average static search cost is approximately $D/2$, where D is the length of the search list. We might expect better results for this data, however, because the distribution of request frequencies in many natural contexts obeys Zipf's Law; the average cost for the Optimal Static Ordering of that distribution is approximately $D/\ln D$ (see Knuth [1973, Section 6.1]). The column in Table 1 entitled "Zipf's Law" gives the cost of the Optimal Static Ordering if the requests had been drawn from that distribution; comparing that column to the cost of the Optimal Static Ordering shows that the data is closer to a Zipf distribution than a uniform distribution.

The files were obtained from user accounts and an on-line documentation system, and were grouped into two classes: Pascal files and Text files. The characteristics of the classes vary, so we consider the results for each separately.

The four Pascal files tested contained between 100 and 181 distinct words (corresponding to the length of the key list), which were requested a total of 431 to 1456 times (corresponding to the length of the request sequence). The empirical results for the Pascal files were striking: Move-to-Front and Count performed dramatically better than Transpose, and in two cases, Move-to-Front required fewer comparisons than the Optimal Static Ordering. The high *locality* present in source code accounts for this surprising phenomenon: infrequently used words such as integer appear in groups rather than being uniformly distributed throughout the file. Where a request for such a word would require a long search under the Optimal Static Ordering, the search under Move-to-Front would be short after the first request, since the key would then be at the beginning of the list. The Count heuristic can also exploit the locality of keywords such as **real** and **integer**; at the beginning of the program text, their counts will be higher than the counts of other words. For Transpose, the requested element may not have time to drift towards the front and high-locality words that occur in the same neighborhood can contend with one another, so the search remains expensive. This phenomenon of locality in such constructs as **Total := Total + 1**, **end; end; end**, and **var** declarations enhances the performance of Move-to-Front considerably.

The Text files included the text of the Constitution of the United States (T6 in Table 1), the script to *The Rocky Horror Picture Show* (T5), a version of this paper (T4), excerpts from an on-line documentation system, and text files augmented with instructions to the Scribe document production system. While Transpose still required more comparisons than the other two heuristics, its performance was better for this class of files. Move-to-Front performed best in most cases, although it never beat the Optimal Static Ordering. The Count heuristic was never far behind Move-to-Front, and in two cases performed better than Move-to-Front.

	Distinct Words	Total Words	Zipf's Law	Optimal Static Ordering	Move-to Front	Count	Transpose
Pascal Files							
P 1	100	480	18.28	27.52	24.49	33.16	40.43
P 2	107	431	19.36	26.23	25.62	31.50	38.92
P 3	117	1,176	20.90	18.04	18.21	20.63	30.53
P 4	181	1,456	30.32	30.78	31.40	35.71	47.41
Text Files							
T 1	471	1,888	68.95	93.03	104.46	111.21	147.41
T 2	498	1,515	72.36	112.86	119.31	135.63	160.69
T 3	564	3,296	80.58	96.29	98.90	112.41	155.17
T 4	999	5,443	132.48	149.34	168.79	175.42	258.20
T 5	1,147	7,482	149.47	143.72	174.50	166.10	204.74
T 6	1,590	7,654	199.02	232.53	280.83	267.64	349.94

Table 1. Average Search Costs.

The empirical results indicate that neither the worst-case nor the probabilistic analyses by themselves completely describe the behavior of the heuristics under natural conditions: Transpose clearly is not the best heuristic for this application, yet it never performed as badly as our results showed it might. Certainly, the distribution and size of the request sequence seem less significant than the ordering of the requests. Empirical results for the Pascal files would be less dramatic if Pascal reserved words were treated differently from identifiers, as might happen within a compiler. If sequential search were used by an interpreter for identifier lookup at runtime, however, the presence of *dynamic locality* (for example, in requests for loop variables) would argue strongly for the use of Move-to-Front.[4] This phenomenon is not restricted to source code; for example, the word "president" appears with high locality in the U.S. Constitution. In most written prose, locality of subject (and therefore of certain words) determines paragraph construction. Move-to-Front is able to take advantage of this characteristic. Indeed, such a phenomenon as "pairwise-locality" (the appearance of word pairs or two-word phrases such as "vice-president" or A[I]) might hamper the performance of Transpose; if two such words have the misfortune to be adjacent in the key list, then they will contend with each other rather than drifting to the front as they should.

5. Advice to Practitioners

The previous sections have evaluated the heuristics from various viewpoints, using theoretical tools as well as test results for several data sets. We now consider the heuristics from a very different perspective: how should they be used by practicing programmers?

The purpose of the heuristics is to increase the performance of a linear search. This raises the most important point of this section: if a programmer faces an efficiency problem in a search procedure, then linear search is probably not the method of choice. Knuth [1974, Chapter 6] describes a number of other search methods that are usually significantly more efficient. There are, however, contexts in which self-organizing linear search may be appropriate.

- When N is very small (say, at most several dozen), the greater constant factors in the runtimes of other strategies may make linear search competitive. This occurs, for example, when linked lists are used to resolve collisions in a hashing structure.

- When space is severely limited, sophisticated data structures may be too space-expensive to use.

- If the performance of linear search is almost (but not quite) good enough, a self-organizing heuristic may make it effective for the application at hand without adding more than a few lines of code.

[4] The widely-used Microsoft BASIC interpreter stores symbols in its run-time (linear) symbol table in the order in which they were first seen. One of the authors (JLB) once reduced the run time of a production BASIC program under such an interpreter from fourteen hours to seven hours simply by referring to each "hot" variable once in a dummy statement at the front of the program. The use of the Move-To-Front heuristic in such an interpreter would probably substantially decrease the run time of many BASIC programs.

Ed McCreight [1983] found himself in the last situation when improving the performance of a VLSI circuit simulator that had two primary phases: the first phase read the description of the circuit and the second phase then simulated the circuit. On typical runs the first phase would take five minutes while the second phase would take several hours. Although the five minutes of the first phase was not crucial, it was irritating for users to have to wait that long to see the simulation begin (especially when they knew that most of the time was going to sequential searches in the simulator's symbol table). Following McCreight's suggestion, the implementer of the program augmented the straightforward sequential search with the Move-to-Front heuristic. Those additional half-dozen lines of code decreased the runtime of the first phase from five minutes to half a minute (most of which was *not* going to symbol table routines).

The lesson to be learned from the above paragraph is that when efficiency matters in a search routine, then non-linear data structures (especially hashing) should be seriously considered. Sometimes, however, self-organizing heuristics can be exactly the right tool for the job by providing enough runtime efficiency with little overhead in code development.

Knowing when to use self-organization heuristics still leaves the implementer with the decision of which one to choose in a given application. Some authors have interpreted the results in Section 2 in a way we feel is unwarranted: for instance, Gotlieb and Gotlieb [1978, p. 118] assert in their excellent data structures text that "[Move-to-Front] is not the best [strategy] for a self-organizing list. It is better to promote the referenced entry only one place by transposing it with its predecessor." The following discussion of the heuristics is relevant to most situations in which self-organizing schemes are applicable.

- *Move-to-Front.* The linked list implementation of this heuristic is the method of choice for most applications. The heuristic makes few comparisons, both in the worst case and when observed on real data; furthermore, it exploits any locality of reference present in the input. The linked list implementation is natural for an environment supporting dynamic storage allocation and yields an efficient reorganization strategy. Unfortunately, moving to front is expensive if the sequence is implemented as an array.

- *Transpose.* If storage is extremely limited and pointers for lists cannot be used, then the array implementation of Transpose gives very efficient reorganization. Its worst-case number of comparisons is high, but it performs well on the average.

- *Count.* Although this heuristic does make a small number of comparisons in the worst case, its extra storage and higher move costs make it unattractive for most applications. It should probably be considered only in applications in which the counts are already needed for other purposes.

In the above discussion we have intentionally kept vague several potentially quantifiable measures. Rather, we appeal to an intuition that asymptotically efficient algorithms tend to require more code and to have larger constant factors. We have avoided hard data because it is extremely sensitive in this context to coding style and to details of the compiler and machine architecture. Readers who insist on such detail should consult Knuth [1973, Chapter 6], but we warn that data on his MIX implementations may be misleading for other computing environments.

6. Conclusions

The conclusions of this paper are clear: when a self-organizing sequential search is appropriate in an application, the Count and (especially) the Move-To-Front heuristics should be considered for implementation. Although previous probabilistic analyses showed that Transpose is superior to Move-To-Front under some measures, both our worst-case analyses and our empirical results show contexts in which the opposite is true.

The theoretical results in this paper could be extended in a number of ways. An implementer of these algorithms may wish to consider measurements other than number of comparisons, such as number of moves or total distance moved. The worst-case analysis of algorithms previously analyzed only for their expected performance is an interesting open problem. To predict more accurately the behavior of the heuristics on input like that described in the previous section, it would be helpful to have theoretical tools for describing the locality present in the input. Also, the proof techniques that we presented could be used to study other self-modifying structures in the worst-case sense.[5]

Acknowledgements

The helpful comments of Al Aho, Jim Saxe and Bruce Weide are gratefully acknowledged.

References

Anderson, E. J., P. Nash, and R. R. Weber [1982]. A counterexample to a conjecture on optimal list ordering. *J. Appl. Prob.*, to appear.

Bitner, J. R. [1976]. *Heuristics that dynamically alter data structures to reduce their access time.* Ph.D Thesis, University of Illinois, Urbana-Champaign, IL.

Bitner, J. R. [1979]. Heuristics that dynamically organize data structures. *SIAM J. Comp. 8*, 1 (February 1979), 82-110.

[5] D. D. Sleator and R. E. Tarjan of Bell Laboratories have used techniques fundamentally different from those in this paper to prove several of the results in this paper and many other new results dealing with the worst-case performance of sequential search self-organization heuristics; their results will be described in their forthcoming paper, "Amortized efficiency of list update and paging rules."(Personal communication to JLB.)

Bitner, J. R. [1982] Two results on self-organizing data structures. Department of Computer Science, University of Texas at Austin, Austin, TX, TR-189 (January 1982).

Burville, P. J. and J. F. C. Kingman [1973]. On a model for storage and search. *J. Appl. Prob. 10,* (September 1973), 697-701.

Gonnet, G., J. I. Munro, and H. Suwanda [1979]. Toward self-organizing sequential search heuristics. *Proc. 20th IEEE Symp. Foundations Computer Science,* 169-174.

Gotlieb, C. C. and L. R. Gotlieb [1978]. *Data Types and Structures.* Prentiss-Hall Englewood Press, New Jersey.

Hendricks, W. J. [1972]. The stationary distribution of an interesting Markov chain. *J. Appl. Prob. 9,* (March 1972), 231-233.

Hendricks, W. J. [1973]. An extension of a theorem concerning an interesting Markov chain. *J. Appl. Prob. 10,* (December 1973), 886-890.

Hendricks, W. J. [1976]. An account of self-organizing systems. *SIAM J. Comp. 5,* 4 (December 1976), 715-723.

Kan, Y. C. and S. M. Ross [1980]. Optimal list order under partial memory constraints. *J. Appl. Prob. 17,* (December 1980), 1004-1015.

Knuth, D. E. [1973]. *The Art of Computer Programming, Vol 3: Sorting and Searching.* Addison-Wesley, Reading, MA.

Lam, K., M. K. Sui, and C. T. Yu [1981]. A generalized counter scheme. *Theoretical Comp. Sci. 16,* (December 1981), 271-278.

McCabe, J. [1965]. On serial files with relocatable records. *Oper. Res. 12,* (July 1965), 609-618.

McCreight, E. [1983]. Personal communication with Ed McCreight of Xerox Palo Alto Research Center to JLB, March 1983.

Rivest, R. [1976]. On self-organizing sequential search heuristics. *CACM 19,* 2 (February 1976), 63-67.

Schay, G. Jr., and F. W. Dauer [1967]. A probabilistic model of a self-organizing file system. *SIAM J. Appl. Math. 15,* (February 1967), 874-888.

Tenenbaum, A. [1978]. Simulations of dynamic sequential search algorithms. *CACM 21,* 9 (September 1978), 790-791.

Tenenbaum, A. and R. M. Nemes [1982]. Two spectra of self-organizing sequential search algorithms. *SIAM J. Comp.*, to appear.

Veinott, A. F. [1965]. Optimal policy in a dynamic, single product, nonstationary inventory mode with several demand classes. *Oper. Res. 13*, 761-778.